Executing Hierarchical Interactive Scores in ReactiveML

Jaime Arias, Myriam Desainte-Catherine, Sylvain Salvati, Camilo Rueda

EXECUTING HIERARCHICAL INTERACTIVE SCORES IN REACTIVEML

Jaime Arias, Myriam Desainte-Catherine, Sylvain Salvati
Univ. Bordeaux, LaBRI, Bordeaux, F-33000, France
CNRS, UMR 5800, Bordeaux, F-33000, France
IPB, LaBRI, Bordeaux, F-33000, France
{jaime.arias, myriam, sylvain.salvati}@labri.fr

Camilo Rueda
Departamento de Electrónica y
Ciencias de la Computación
Pontificia Universidad Javeriana
Cali, Colombia
crueda@javerianacali.edu.co

RÉSUMÉ

Le modèle des partitions interactives permet d'écrire et d'exécuter des scénarios multimédia interactifs. Le logiciel I-SCORE implémente ce modèle au moyen des Hierarchical Time Stream Petri Nets (HTSPN). Cependant, cette implémentation est très statique et l'ajout de certaines fonctionnalités peut nécessiter une reconception complète du réseau. Un autre problème de I-SCORE est qu'il ne fournit pas un bon retour visuel de l'exécution d'un scénario. Dans cet article, nous définissons et implémentons un interprète de partitions interactives avec le langage de programmation synchrone REACTIVEML. Dans ce travail, nous tirons parti de l'expressivité du modèle réactif et de la puissance de la programmation fonctionnelle pour obtenir un interprète plus simple et plus dynamique. Contrairement à l'implémentation basée sur les réseaux de Petri, cette approche permet de définir précisément l'aspect hiérarchique et permet de prototyper facilement de nouvelles fonctionnalités. Nous proposons aussi une visualisation temps réel de l'exécution en utilisant l'environnement INSCORE.

ABSTRACT

Interactive scores proposes a model to write and execute interactive multimedia scores. The software I-SCORE implements the above model using Hierarchical Time Stream Petri Nets (HTSPN). However, this model is very static and modelling new features would require a complete redesign of the network or sometimes they cannot be expressed. Another problem of I-SCORE is that it does not provide a good visual feedback of the execution of the scenario. In this work, we define and implement an interpreter of interactive scores using the synchronous programming language REACTIVEML. Our work takes advantage of the expressiveness of the reactive model and the power of functional programming to develop an interpreter more dynamic and simple. Contrary to the Petri Net model, our approach allows to model precisely the hierarchical behaviour, and permits the easy prototyping of new features. We also propose a visualization system using the environment INSCORE that provides a real-time visualization of the execution of the score.

1. INTRODUCTION

Interactive scores [10] proposes a model to write and execute interactive scenarios composed of several multimedia processes. In this model, the temporal organization of the scenario is described by means of flexible and fixed temporal relations among temporal objects (i.e., multimedia processes) that are preserved during the writing and performance stage.

The implementation of interactive scores in the software I-SCORE [1] is based on Petri Nets [17]. Such implementation provides an efficient and safe execution, but implies a quite static structure. Indeed, only elements that have been planned during the composition process can be executed. Therefore, it is not possible to modify the structure of the scenario during execution, for example, dynamically add a new element that was not written before execution. In addition, modelling new features for I-SCORE such as conditionals, loops or handling streams would require a complete redesign of the network. Therefore, this model is not suitable for compositional development and integration of new features that composers increasingly need to write more complex scenarios.

In this paper, we explore a new way to define and implement interactive scores, aiming at a more dynamic model. For this purpose, we use REACTIVEML [16], a programming language for implementing interactive systems (e.g., video games and graphical user interfaces). This language is based on the synchronous reactive model of Boussinot [8], then it provides a global discrete model of time, clear semantics, and unlike Petri nets, synchronous and deterministic parallel composition and features such as dynamic creation of processes. Moreover, REACTIVEML has been previously used in music applications [4, 5] showing to be very expressive, efficient, capable of interacting with the environment during the performance of complex scores, and well suited for building prototypes easily.

The rest of the paper is organized as follows. In Section 2 we present the I-SCORE system and we briefly introduce

[1] http://i-score.org

the REACTIVEML programming language. In Section 3 we describe the implementation in REACTIVEML of the new interpreter of interactive scores. Next, in Section 4 we present the improved visualization system in INSCORE. Finally, in Section 5 we present related work, conclusions, and ideas for future work.

2. PRELIMINARIES

In this section we present the I-SCORE system and the necessary notions of REACTIVEML language.

2.1. I-SCORE

I-SCORE [3, 17] is a software for composing and executing interactive multimedia scenarios [2]. It consists of two sides: *authoring* and *performance*. In the authoring side, the composer designs the multimedia scenario while in the performance side the performer executes the scenario with interactive capabilities. Next, we present in more detail both sides.

2.1.1. Authoring side

In interactive scores [10], multimedia elements are temporal structures represented as boxes. These boxes can be either *simple* or *complex*. A simple box represents a multimedia process that will be executed by an external application such as PURE DATA [2] or MAX/MSP [3]. I-SCORE controls such applications by means of OSC [4] messages. On the other hand, a complex box allows to gather and execute a set of boxes making possible the design of large and complex scenarios. However, this box does not execute a multimedia process.

The temporal organization of the score is partially defined by temporal relations. They indicate a precedence relation between boxes using Allen's relations [1], as well as a delay between them. A temporal relation can be either *rigid* or *flexible*. In a rigid relation, the duration of the delay is fixed whereas in a flexible relation, the duration is partially defined by an interval of time (i.e., it has a minimum and a maximum duration).

The composer can add interaction points to boxes. They allow to modify the preceding relations (i.e., start date) and the duration (i.e., end date) of boxes during the execution. In the case of a complex box, an interaction point stops abruptly the box with its children. It is important to know that the temporal organization of the score is preserved during composition and performance. Therefore, the performer can interpret the same score in different ways within the constraints of the composer. In I-SCORE, an interaction point is triggered by sending the OSC message defined by the composer. Additionally, all preceding relations of a box with an interaction point are flexible, otherwise they are rigid.

[2] http://puredata.info
[3] http://cycling74.com/products/max/
[4] http://opensoundcontrol.org/introduction-osc

In Figure 1 we illustrate an interactive scenario with all its temporal relations. Here, the horizontal axis represents time and the vertical axis has no meaning. Additionally, the start and end of boxes are partially defined by temporal relations that are represented as solid (rigid relation) and dashed (flexible relation) arrows. Moreover, tabs represent the interactions points of boxes.

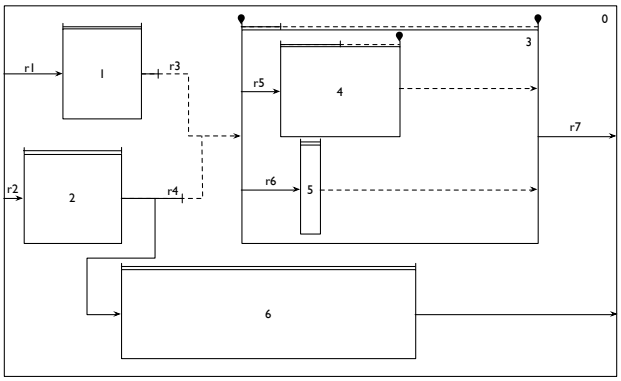


Figure 1. Example of an interactive scenario.

2.1.2. Performance side

In order to execute the scenario designed in the previous side, I-SCORE translates the score into a Hierarchical Time Stream Petri Net (HTSPN) [19]. The Petri net model allows to trigger the interactive events, and also it denotes and preserves the temporal organization of the score during the execution. The reader may refer to [17] for further information on generating the HTSPN structure from the score. The following example illustrates the execution of a scenario.

Example 1. Consider the interactive score in Figure 1 with the following configuration :

- Box 1 starts 3 seconds after the start of the scenario. Its duration is 4 seconds.
- Box 2 starts 1 second after the start of the scenario. Its duration is 5 seconds.
- Box 6 starts immediately (i.e., 0 seconds) after the end of the box 2. Its duration is 15 seconds.
- Box 3 is a complex box with two children; the box 4 and 5. The start of the box is defined by the flexible relations $r3$ and $r4$ whose durations are $[1, 4]$ and $[3, 8]$, respectively. Therefore, at any instant in which the above relations are satisfied, the box may be started by triggering the interaction point. In Section 3 we elaborate more on the semantics of temporal relations and interaction points.
- The duration of the box 3 is the interval $[2, \infty]$, then the box may be stopped after 2 seconds of its starting by triggering the interaction point.

- Box 4 starts 2 seconds after the start of its parent (i.e., box 3). Its duration is the interval $[3, 6]$, then the box may be stopped after 3 seconds of its starting by triggering the interaction point. It is important to know that the box will stop when it reaches its maximum duration (i.e., 6 seconds) if the interaction point is not triggered before.
- Box 5 starts 3 seconds after the start of the box 3. Its duration is 1 second.
- The scenario finishes when the box 6 finishes and 4 seconds have elapsed after the end of the box 3.

Following [17], we translated the score into its equivalent HTSPN structure (Figure 2). For the sake of simplicity, we do not show the time interval on the arcs. Note that transitions with double stroke are those with an interaction point.

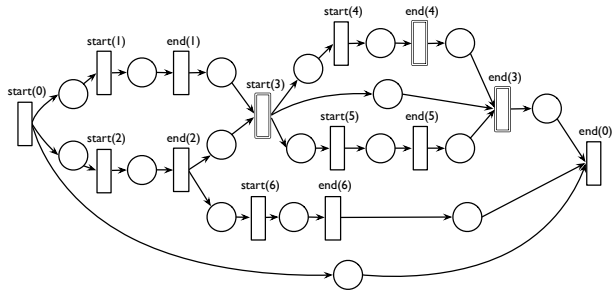


Figure 2. HTSPN structure of the scenario specified in Example 1.

We show in Figure 4 an execution of the above scenario where only the interaction point of the box 3 is triggered. All other intervals reach their maximum duration. Note that box 3 must be stopped, otherwise it will never end because it has an infinite duration. `merge(r3,r4)` represents the interval of time in which the box 3 may start. In this interval, the relations $r3$ and $r4$ are satisfied. As can be seen, the interaction point is not triggered, then the box starts when the interval reachs its maximum duration. We can conclude that this scenario finishes in 27 seconds only if the interaction point at the end of the box 3 is triggered at 23 seconds.

2.2. REACTIVEML

REACTIVEML [16] is a synchronous reactive programming language designed to implement interactive systems such as graphical user interfaces and video games. It is based on the reactive model of Boussinot [8] and it is built as an extension of the functional programming language OCAML [5]. Therefore, it combines the power of functional programming with the expressiveness of synchronous paradigm [6].

The reactive synchronous model provides the notion of a global logical time. Then, time is viewed as a sequence of logical instants. Additionally, parallel processes are executed synchronously (*lock step*) and they communicate with each other in zero time. This communication is made by broadcasting signals that are characterized by a status defined at every logical instance: *present* or *absent*. In contrast to ESTEREL [7], the reaction to absence of signals is delayed, then the programs are causal by construction (i.e., a signal cannot be present and absent during the same instant). Moreover, the reactive model provides dynamic features such as dynamic creation of processes. Indeed, REACTIVEML provides a toplevel [15] to dynamically write, load and execute programs.

In REACTIVEML, regular OCAML functions are instantaneous (i.e., the output is returned in the same instant) whereas *processes* (**process** keyword) can be executed through several instants. Next, we use the program shown in Figure 3 to describe the basic expressions of REACTIVEML.

```
1  let process killable_p p s =
2    do
3      run p
4    until s done
5
6  let process wait tic dur =
7    for i=1 to dur do await tic done
8
9  let process emit_tic period tic =
10 let start = Unix.gettimeofday () in
11 let next = ref (start +. period) in
12 loop
13   let current = Unix.gettimeofday() in
14     if (current >= !next) then begin
15       emit tic ();
16       next := !next +. period
17     end;
18     pause
19 end
```

Figure 3. Example of REACTIVEML language.

Two expressions can be evaluated in sequence (`e1;e2`) or in parallel (`e1||e2`). In REACTIVEML is possible to write higher order processes like the process `killable_p` (line 1) which takes two arguments: a process `p` and a signal `s`. This process executes `p` until `s` is present. The expression **run** executes a process (line 3). There are two important control structures: the construction **do** `e` **until** `s` to interrupt the execution of `e` when the signal `s` is present, and the construction **do** `e` **when** `s` to suspend the execution of `e` when the signal `s` is absent.

Signals can be emitted (**emit**), and awaited (**await**). For instance, the process `wait` (line 6) takes two arguments: a signal `tic` and an integer `dur`. The purpose of this process is similar to a timer; it waits for the signal `tic` to be emitted a number `dur` of times. The expression **await** `s` waits for `s` to be emitted and it finishes in the next instant whereas the expression **await immediate** `s` waits for `s` to be emitted and it terminates instantaneously. An important characteristic of the REACTIVEML implementation is the absence of busy waiting: nothing is computed when no signal is present. The process `emit_tic` (line 9) takes two arguments: a float `period` and a signal `tic`. It works like a clock; it gets the current time by using the

[5] `http://ocaml.org`

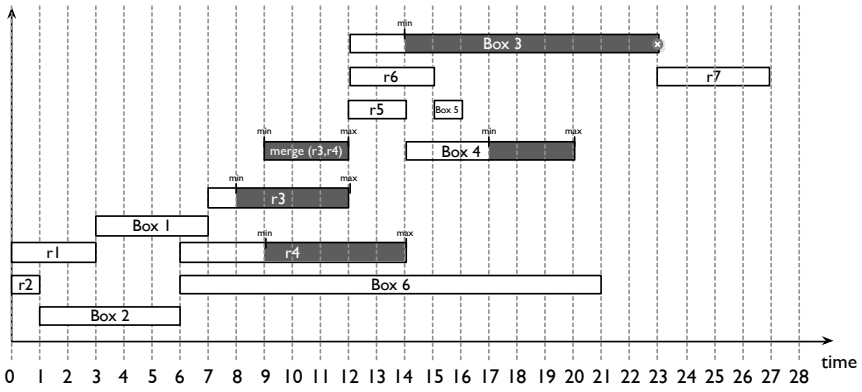


Figure 4. Execution of Example 1 where only the interaction point of the box 3 is triggered.

function `Unix.gettimeofday` from the `Unix` module, and emits the signal `tic` (line 15) whenever the period of time expires (line 14). The **`pause`** (line 18) keyword awaits for the next instant. The construction **`loop`** `e` **`end`** iterates infinitely `e`.

REACTIVEML also provides valued signals. They can be emitted (**`emit`** `s value`) and awaited to get the associated value (**`await`** `s (pattern)`**`in`** `expression`). Different values can be emitted during an instant (*multi-emission*). In this case, it is necessary to define how the emitted values will be combined during the same instant (**`signal`** `name` **`default`** `value` **`gather`** `function` **`in`** `expression`). The value obtained is available at the following instant in order to avoid causality problems. For example, the process `add` (Figure 5) declares the local signal `num` (line 2) with an initial value `0` and a function which adds two integers. The process `gen` (line 3) generates a set of values that are emitted through the signal `num` at the same instant. The process `print` (line 6) awaits for the signal `num`, and then it prints the value in `n`. Note that `n` contains the sum of all values generated by the process `gen`.

```
1  let process add max =
2    signal num default 0 gather fun x y -> x+y in
3    let process gen =
4      (for i=1 to max do emit num i done)
5    in
6    let process print =
7      await num (n) in
8      print_endline (string_of_int n)
9    in
10     run gen || run print
```

Figure 5. Example of multi-emission of signals.

3. SYNCHRONOUS MODEL OF INTERACTIVE SCORES

In this section, we present a new execution model of interactive scores using the reactive programming language REACTIVEML. The implementation of the interpreter is divided into two main modules: `Time` and `Motor`. The module `Time` interfaces the abstract time relative to the tempo (in beats) and the physical time (in ms). This module is based on the work of Baudart *et al.* [4, 5]. The module `Motor` interprets the interactive score and interacts with the environment by listening external events and triggering external multimedia processes. The implementation fulfills the operational semantics of interactive scores [10]. In the following, we describe the above modules.

3.1. Modelling Time

REACTIVEML, like other synchronous languages, provides the notion of a global logical time. Then, time is viewed as a sequence of logical instants. The process `emit_tic` (Figure 3), explained in Section 2.2, is the interface between the physical time and the logical time. Its purpose is to generate the clock of the system by emitting a signal in a periodic time. Therefore, from this signal of clock, we can define a process to express delays by waiting a specific number of ticks (process `wait` in Figure 3).

3.2. Execution of Interactive Scores

Interactive scores [10] are basically composed of: boxes that represent multimedia processes; temporal relations that define the temporal organization of the score (i.e., the start and the end of boxes); and interaction points that transform a static score into dynamic by allowing the performer to modify the temporal relations during the execution. In the following, we present the implementation of the above elements in REACTIVEML.

3.2.1. Temporal relations

Temporal relations partially define the temporal organization of the scenario. They represent delays that allow to specify the start and the duration of boxes. Relations can be either *rigid* or *flexible*. In a rigid relation, the duration of the delay is fixed whereas in a flexible relation, the duration of the delay is partially defined by an interval of time. This interval can be modified by triggering an *interaction point* during the execution.

Hence, we define two types of intervals in order to represent temporal relations. The *fixed interval* represents a rigid relation and the *interactive interval* represents a flexible relation with an attached interaction point (Figure 6). The interaction point allows to stop the interval during the defined interval of time (i.e., between the minimum and the maximum duration) by triggering a specific event.

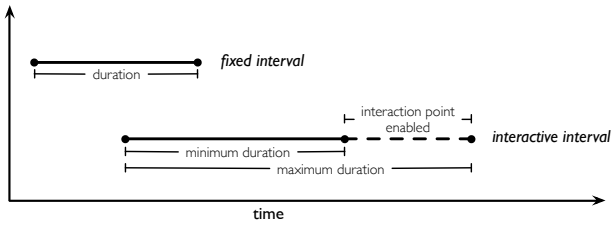


Figure 6. Fixed and flexible intervals.

In REACTIVEML, we represent a fixed interval as a tuple `(d,s)` where the signal `s` is emitted when the duration `d` has elapsed. On the other hand, the interactive interval is defined as a tuple `(min,max,ip)` where `min` and `max` are fixed intervals that represent the minimum and maximum duration of the interval, and `ip` is its interaction point. It should be noted that the duration of a flexible interval can be either finite or infinite.

Interaction points are represented as OSC messages that are sent from the environment and transmitted through a signal. An OSC message is represented as a tuple `(t, a)` where `t` is the address and `a` is the list of arguments with their type. For example, `('/light/1', [String ' luminosity'; Int32 90])`.

Hence, we define a temporal relation between two boxes as a tuple `(from,to,intrvl)` where `from` and `to` are the identifiers of the boxes involved in the relation, and `intrvl` is the interval that defines the delay between them.

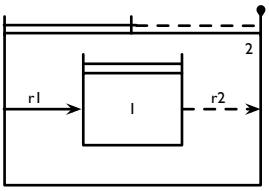
3.2.2. Boxes

Boxes can be either *simple* or *complex*. A simple box represents a multimedia process that is executed by an external application. For this reason, the interpreter only sends OSC messages in order to control its start and end. On the other hand, a complex box gather and execute a set of boxes with their own temporal organization. Recall that the duration of a box is defined by an interval. Then, it can be either fixed or interactive. In the rest of the paper, we call rigid boxes as process boxes and complex boxes as hierarchical boxes.

A process box is defined as a tuple `(id,intrvl,s_msg ,e_msg)` where: `id` is the identifier of the box; `intrvl` is the interval that defines its duration; `s_msg` and `e_msg` are the OSC messages to start and stop the external process, respectively. We define a hierarchical box as a tuple `(id, b_list, r_list, intrvl)` where: `id` is the identifier of the box; `b_list` is the list of its children; `r_list` is the list of the relations; and `intrvl` is the interval that defines its duration. The following example illustrates the representation of boxes and relations in REACTIVEML.

Example 2. Consider the hierarchical box shown in Figure 7a. This box, identified with the number 2, has a process box as child and an interaction point at the end. The duration of the hierarchical box is the interval $[2, \infty]$. The process box, identified with the number 1, starts 2 seconds after the starting of its parent and its duration is 2 seconds.

We show the specification in REACTIVEML of Example 2 in Figure 7b.



(a) Graphical representation.

```
1 signal pb1_s, hb1_s, hb2_s, r1_s, r2_s;
2 let ip = ("/stop",[Int32 3]);
3 let start_m = ("/box/1",[String "start"]);
4 let stop_m = ("/box/1",[String "stop"]);
5 let r1 = Fixed (Finite 2, r1_s);
6 let r2 = Fixed (Finite 0, r2_s);
7 let p_box = Process (1, Fixed (Finite 2, pb1_s), start_m,
      stop_m);
8 let h_box = Hierarchical (2, [p_box], [(2,1,r1);(1,2,r2)
    ], Interactive ((Finite 2, hb1_s), (Infinite, hb2_s)
    , ip))
```

(b) Specification in REACTIVEML.

Figure 7. Specification of a hierarchical box in REACTIVEML.

Firstly, we define the signals that will be emitted when the intervals reach their durations (line 1), the OSC message of the interaction point (line 2), and the OSC messages to start (line 3) and stop (line 4) the external multimedia process. Then, we define the interval `r1` (line 5) that determines the start of the process box (i.e., box 1). It is important to note that we need to specify both the type of the interval (i.e., `Fixed` or `Interactive`) and the duration (i.e., `Finite` or `Infinite`). Since the parent of the box 1 has an interaction point, the duration of the interval `r2` is not relevant, therefore we define its duration as 0 seconds (line 6). Next, we define the process box `p_box` (line 7) with a fixed duration of 2 seconds and the messages defined previously. Finally, we define the hierarchical box with its child `p_box`, its internal relations `r1` and `r2`, and its duration. Note that the duration of box 2 is defined by means of an interactive interval because it has an interaction point at the end. As in the definition intervals, we need to specify the type of the box (i.e., `Process` or `Hierarchical`).

The execution of a box is performed by a REACTIVEML process (Figure 8). It first waits for the preceding intervals of the box are satisfied (line 3). Then, it executes the box depending of its type (line 4). Finally, once the box has finished, it launches its succeeding intervals (line 5). The above processes must be killed if the parent of the box is

stopped. In the following we describe the processes that decode each type of box.

```
1 let rec process run_generic box w_rels s_rels stop_f
      id_box =
2   (* .. *)
3   run (killable_p (wait_intervals w_rels id_box)
        stop_f);
4   run (run_box box);
5   run (killable_p (run_intervals s_rels) stop_f)
```

Figure 8. Process that executes a box.

A process box, implemented in the process `run_p_box` (Figure 9), first gets its identifier (`ident`), the interval that defines its duration (`interval`), and the OSC messages to start (`start_m`) and stop (`end_m`) the external process (line 2). Then, it starts the external process by sending the corresponding OSC message (line 4). Next, it runs the interval of its duration (line 5) and waits until it ends (line 6). Finally, once the box stops, it immediately sends the corresponding OSC message to stop the external process (line 7). The box and its external process finish suddenly if the signal `stop_f` is emitted (**`do/until`** construction) by its parent.

```
1 let process run_p_box p_box =
2   let (ident, interval,star_m,end_m) = p_box in
3   do
4     emit output (star_m);
5     (run (run_intervals [interval]) ||
6     run (wait_intervals [interval] ident));
7     emit output (end_m);
8   until stop_f -> emit output (end_m) done
```

Figure 9. Process that executes a process box.

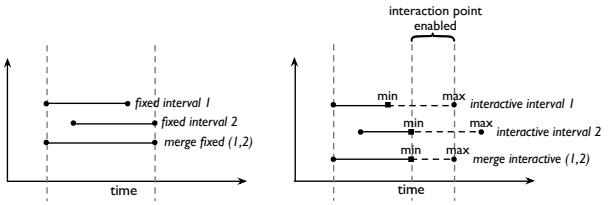
On the other hand, a hierarchical box is executed by the process `run_h_box` (Figure 10). It first gets the parameters of the box (line 2): its identifier (`ident`); its children (`boxes`); the temporal relations of the sub-scenario (`relations`); and the interval that defines its duration (`interval`). Then, it executes in parallel: a monitor that emits the signal `stop_box_h` when the parent of the box finishes suddenly (line 5); the interval that defines its duration (line 7); a monitor that emits the signal `stop_box_h` when the box stops because of an interaction point (line 12); the relations that describe the temporal organization of the sub-scenario (line 15); a monitor that waits for the relations defining its end (line 16); and its children with their preceding and succeeding intervals (line 18). Hence, the hierarchical box and its children will finish abruptly when the signals `stop_box_h` or `stop_f` are emitted. Otherwise, the hierarchical box will finish when its duration and all internal relations have finished.

3.2.3. Synchronization

Boxes can have one or more preceding and succeeding relations. In I-SCORE, all preceding relations of a box with an interaction point are flexible (interactive intervals). Otherwise, all are rigid (fixed intervals). In the first case, the box will start when one of its preceding relations has finished, and the interaction point will be enabled when they have reached their minimum duration (Figure 11b). In the second case, the box will start when all its preceding relations have finished (Figure 11a). As noted, we can merge a set of intervals into one that follows the behaviour described above. Next, we describe the processes to handle intervals.

```
1  let process run_h_box h_box =
2    let (ident, boxes, relations, interval) = h_box in
3    signal stop_box_h in
4    signal kill_m in
5    do (await immediate stop_f; emit stop_box_h) until
         kill_m done ||
6    (((do
7      run (run_intervals [interval]) ||
8      (run (wait_intervals [interval] ident);
9        begin
10         match interval with
11         | Fixed _ -> ()
12         | Interactive _ -> emit stop_box_h
13       end
14     ) ||
15     run (run_intervals (get_intervals ident relations
           From)) ||
16     run (wait_intervals (get_intervals ident relations
           To) ident)
17   until stop_box_h done); emit stop_box_h) ||
18   run (run_boxes_par boxes relations stop_box_h));
       emit kill_m
```

Figure 10. Process that executes a hierarchical box.



(a) Fixed intervals. (b) Interactive intervals.

Figure 11. Merging a set of intervals.

The process `run_intervals` (Figure 12) runs in parallel a list of intervals (line 3). Each interval emits a specific signal when it reaches its duration (line 7). In the case of an interactive interval, a different signal will be emitted when it reaches its minimum and maximum duration (line 12 and 14). Following the semantics described above, if a box has several preceding intervals, it will start when one of them finishes (**`do/until`** construction). We use the process iterator **`Rml_list`**`.par_iter` to execute in parallel a list of processes.

The process `wait_intervals` (Figure 13) waits for a set of intervals are satisfied. Then, in the case that all intervals are fixed, it waits until all intervals end (line 5 and 7). Otherwise, it first waits until all reach their minimum duration (line 5), and then it begins to listen the external events (line 13) until one interval reaches its maximum duration (**`do/until`** construction) or the event associated to the interaction point is triggered (line 14). The emission of the signal `max_s` also will stop all intervals.

```
let process run_intervals inter_l =
  (* .. *)
  run (Rml_list.par_iter
    (proc i ->
      match i with
        | Fixed (d,s) ->
          run (handle_duration d s)
        | Interactive (min,max,_) ->
          let (min_d,min_s) = min in
          let (max_d,max_s) = max in
          begin
            run (handle_duration min_d min_s);
            do
              run (handle_duration max_d max_s)
            until max_s done;
          end
    ) inter_l )
```

Figure 12. Process that runs a set of intervals.

```
let process wait_intervals inter_l id_box =
  (* .. *)
  if (List.length inter_l > 0) then
  begin
    run (sync_minimum inter_l);
    match (List.hd inter_l) with
    | Fixed (_) ->  ()
    | Interactive (_,max,ip) ->
      let (_,max_s) = max in
      begin
        do
          loop
            await input (ip_e) in
            (if (checkIP ip ip_e) then emit max_s);
            pause
          end
        until max_s done
      end
  end
```

Figure 13. Process that waits for a set of intervals.

3.3. Running an Example

In the following, we present two different executions of the scenario specified in Example 1 (Section 2.1) using our interpreter. In both executions the period of the clock was one second. We used PURE DATA to run the multimedia processes and trigger the interaction points by sending OSC messages.

In the first execution we only triggered at 23 seconds the interaction point at the start of the box 3. Comparing the log of execution (Figure 14) with the execution shown in Figure 4, we observe our implementation follows correctly the operational semantics of interactive scores. Recall that Figure 4 illustrates the execution of Example 1 under the same conditions.

On the other hand, in the second execution we started and stopped the box 3 at 10 and 17 seconds, respectively. We illustrate the execution of Example 1 under the above conditions in Figure 16. Note that the box 3 started early because the interaction point was triggered at 10 seconds. Furthermore, the children of the box 3 were stopped abruptly at 17 seconds because the parent was stopped by the interaction point. Comparing the log of execution (Figure 15) with the execution shown in Figure 16, we observe our implementation follows correctly the operational semantics of interactive scores.

```
Simulation log ...
==================

clock 0 -> (scenario started), (h_box 0 started).
clock 1 -> (p_box 2 started).
clock 2 -> .
clock 3 -> (p_box 1 started).
clock 4, 5 -> .
clock 6 -> (p_box 2 finished), (p_box 6 started).
clock 7 -> (p_box 1 finished).
clock 8 -> .
clock 9 -> (start listening ip 1).
clock 10, 11 -> .
clock 12 -> (stop listening ip 2), (h_box 3 started).
clock 13 -> .
clock 14 -> (p_box 4 started), (start listening ip 2).
clock 15 -> (p_box 5 started).
clock 16 -> (p_box 5 finished).
clock 17 -> (start listening ip 3).
clock 18, 19 -> .
clock 20 -> (stop listening ip 3), (p_box 4 finished).
clock 21 -> (p_box 6 finished).
clock 22 -> .
clock 23 -> (event ip 2 triggered), (stop listening ip
      2), (h_box 3 finished).
clock 24, 25 26 -> .
clock 27 -> (h_box 0 finished), (scenario finished).
```

Figure 14. Execution log of Example 1 where only the interaction point at the start of the box 3 was triggered.

```
Simulation log ...
==================

clock 0 -> (scenario started), (h_box 0 started).
clock 1 -> (p_box 2 started).
clock 2 -> .
clock 3 -> (p_box 1 started).
clock 4, 5 -> .
clock 6 -> (p_box 2 finished), (p_box 6 started).
clock 7 -> (p_box 1 finished).
clock 8 -> .
clock 9 -> (start listening ip 1).
clock 10 -> (event ip 1 triggered), (stop listening ip
      1), (h_box 3 started).
clock 11 -> .
clock 12 -> (p_box 4 started), (start listening ip 2).
clock 13 -> (p_box 5 started).
clock 14 -> (p_box 5 finished).
clock 15 -> (start listening ip 3).
clock 16 -> .
clock 17 -> (event ip 2 triggered), (stop listening ip
      2), (p_box 4 finished), (h_box 3 finished).
clock 18, 19, 20 -> .
clock 21 -> (p_box 6 finished), (h_box 0 finished), (
     scenario finished).
```

Figure 15. Execution log of Example 1 where both interaction points of the box 3 were triggered.

It is important to remark that unlike the Petri Net model presented in [2], our model allows to represent precisely the hierarchical behaviour of boxes. As can be seen in Figure 2, the hierarchical box represented by the transitions `start(3)` and `end(3)` only models the gathering of a set of boxes, but it does not model the forced stopping of its children due to an interaction point.

4. IMPROVING THE VISUALIZATION WITH INSCORE

Currently, the graphical interface of I-SCORE does not support a good feedback in real-time of the dynamic execu-

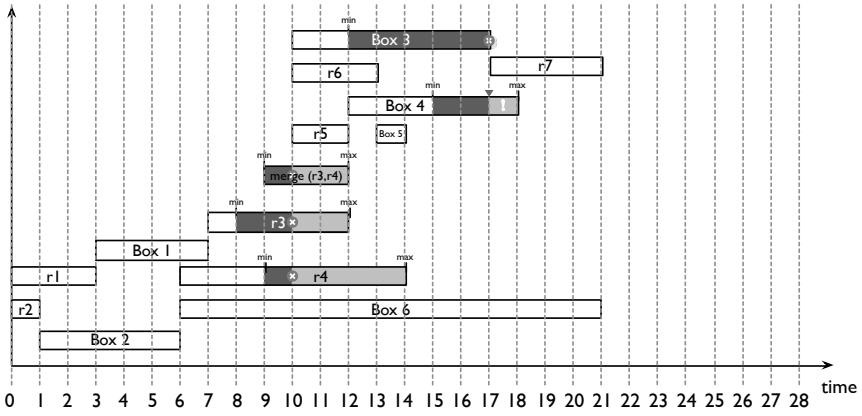


Figure 16. Execution of Example 1 where both interaction points of the box 3 are triggered.

tion of the scenario. In this section, we attempt to overcome this limitation by implementing a graphical interface that changes depending on the events emitted during the execution of a score.

Our approach consists in developing a visualization system in INSCORE [11] that behaves like a synchronous observer [13] of our interpreter (i.e., a process that listens the inputs and outputs of other process without altering its behaviour). INSCORE is a software for designing interactive and augmented scores. Here, scores are composed of heterogeneous graphic objects such as symbolic music notation, text, images, videos and files with a graphic and temporal dimension. Moreover, this tool integrates a message driven system that uses the OSC protocol in order to interact with any OSC application or device. Therefore, the graphical interface can dynamically transform depending on the messages.

Roughly, in our graphical interface (Figure 17) events can be triggered by clicking on the box. Single-click triggers the interaction point at the start of the box whereas double-click triggers the interaction point at the end. The performer knows that an interaction point can be triggered when the border of the box is either dashed (the interaction point at the start) or dotted (the interaction point at the end). A REACTIVEML process listens the events emitted by the interpreter and changes the organization and size of boxes in the graphical interface depending on them. Moreover, boxes change their colour when they are executing. The interface also shows the current time in the upper right of the scenario and indicates the current position of the execution with a vertical line.

4.1. Implementation

In the following we describe the implementation of the process `box`. This process listens the events emitted by the interpreter and dynamically transforms the graphical interface depending on them.

Roughly speaking, the process sends OSC messages to INSCORE according to both the events emitted by the interpreter and the current time of execution. For instance,

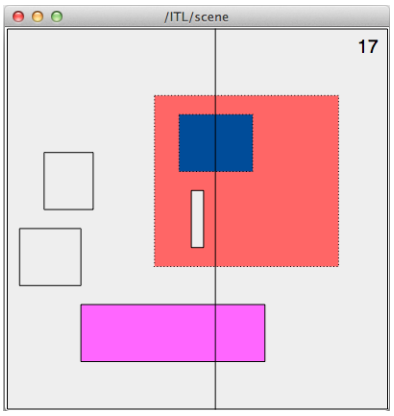


Figure 17. Graphical interface in INSCORE.

a box is moved to the right of the x-axis if the interpreter has not emitted its start event and its start date has elapsed. Additionally, we take advantage of the interaction capabilities of INSCORE to trigger the interaction points of the boxes directly from the graphical interface. Next, we describe in more detail the process `box` (Figure 18).

First, it draws the box in INSCORE by means of the function `draw_box` (line 2). This function also assigns INSCORE events to boxes in order to trigger their interaction points. Next, it verifies at each tick of clock if the box needs to move to the right of the x-axis (line 8) because it has not started and its start date has elapsed (i.e., the start date of the box is delayed). In the case of a hierarchical box, the children move along with the parent (line 10). Additionally, it checks if the interaction point at the start of the box is enabled. If that is true, the box changes its border to dashed (line 16).

Once the box starts (i.e., the interpreter emitted the start event of the box), it changes its colour (line 20) and returns to its original border (line 21). Then, the process verifies if the box started before its start date (line 22). In that case, the box is moved to the current position in the execution of the score (line 24). Next, the process tests at each tick of clock if the width of the box needs to be lengthened (line

33) because it has not stopped and its end date has elapsed (i.e., the end date of the box is delayed). In addition, it examines if the interaction point at the end of the box is enabled. If that is true, the box changes its border to dotted (line 39).

Once the box stops (i.e., the interpreter emitted the stop event of the box), it returns to its original colour (line 43) and border (line 44). Finally, the process verifies if the box stopped before its end date (line 46). In that case, the box is resized to the current position in the execution of the score (line 47). The box with identifier 0 is not resized because we do not want to resize the scenario.

```
1  let process box b =
2    draw_box id_b (!width *. dx) (get_x !pos_x) pos_y
         height ip_start ip_end;
3
4    (* waiting the start *)
5    do
6      loop
7        await immediate clock;
8        if (!pos_x < current) then begin
9          pos_x := !pos_x +. 1.0;
10         List.iter
11           (fun i -> move_box i dx; emit dx_s.(i) 1.0)
12           (id_b::children)
13       end;
14       pause
15     end ||
16     (await immediate start_ip.(id_b); change_border
           id_b "dash")
17   until start_s.(id_b) done;
18
19   (* box started *)
20   change_color id_b r g b ;
21   change_border id_b "solid";
22   if (!pos_x > current) then begin
23     let new_x = current -. !pos_x  in
24     List.iter
25       (fun i -> move_box i (new_x *. dx); emit dx_s.(i
             ) new_x)
26       (id_b::children)
27   end;
28
29   (* waiting the end *)
30   do
31     loop
32       await immediate clock;
33       if (current >= (!pos_x +. !width)) then begin
34         width := !width +. 1.0;
35         resize_box id_b (!width *. dx)
36       end;
37       pause
38     end ||
39     (await immediate start_ip.(id_b); change_border
           id_b "dot")
40   until end_s.(id_b) done;
41
42   (* box stopped *)
43   change_color id_b 238 238 238 ;
44   change_border id_b "solid";
45   let new_dy = (current -. !pos_x) in
46   width := if new_dy > 0.0 then new_dy else 0.0;
47   if (id_b <> 0) then  resize_box id_b (!width *. dx);
```

Figure 18. Process that encodes the behaviour of a box in the graphical interface.

Hence, each box shown in the graphical interface (i.e., INSCORE score) represents a process box that is running concurrently and interacting with other boxes of the graphical interface. Therefore, our visualization system allows performers to observe the current state of the execution of the score.

5. CONCLUDING REMARKS

In this work, we presented a synchronous interpreter of multimedia interactive scores. It was implemented in the synchronous programming language REACTIVEML [16]. We showed that the implementation is simple and small thanks to the synchronous model and high-order programming provided by REACTIVEML. Contrary to the Petri Net model presented in [17], our approach allows to model precisely the hierarchical behaviour of boxes.

We explored the use of INSCORE to develop a graphical interface that provides a real-time visualization of the execution of the score. In this sense, it improves the current graphical interface of I-SCORE. We took advantage of the OSC protocol to communicate our interpreter with external applications such as PURE DATA and INSCORE.

We believe that our implementation provides many advantages for the composition and execution of interactive scores. For instance, as shown in [5], we can prototype new features easily and execute living code using the toplevel of REACTIVEML [15]. Moreover, our approach would allow to execute dynamic processes unlike Petri Nets.

Related work. The work in [4, 5] embeds the ANTESFOCO language [9] and presents how to program mixed music in REACTIVEML. ANTESFOCO is a score following system that synchronizes in real-time electronic music scores with a live musician. The approach defines a synchronous semantics of the core language of ANTESFOCO, and then it is implemented in REACTIVEML. Therefore, composers can prototype new constructs and take advantage of the expressiveness of synchronous model and the power of functional programming. For example, recursion, high order programming, type induction, among others.

Future work. Multimedia interactive scores have a wide range of applications such as video games and museum installations. Therefore, in some cases, it is highly necessary to use conditions and loops in order to model the dynamics of the score easier and correctly. However, these constructions are not supported by the current model of I-SCORE. For this reason, we plan to take advantage of the features of REACTIVEML to prototype these new constructions.

Nowadays, composers have increasingly needed to manipulate streams in their multimedia scenarios. Then, we plan to examine the data-flow programming language LUCID SYNCHRONE [18] in order to handle streams in real-time. This programming language combines the synchronous model of LUSTRE [12] with the features of ML languages. In addition, we plan to perform tests to verify that the new solutions are more efficient than the current implementations.

We intend to increase the usability of our interpreter by developing a compiler that translates automatically scenarios designed in I-SCORE into the syntax of the interpreter. Additionally, we plan to improve the graphical interface in INSCORE in order to provide an environment

where the user can directly design and visualize the execution of a scenario.

Finally, we intend to verify properties of scenarios [14]. For instance, we are interested in knowing the maximum number of processes that can be executed in parallel during all possible executions of the scenario.

Acknowledgements. We thank the anonymous reviewers for their detailed comments that helped us to improve this paper. Also, we would like to thank Louis Mandel for his valuable remarks about the implementation. This work has been supported by the OSSIA (ANR-12-CORD-0024) project and SCRIME [6] .

6. REFERENCES

[1] Allen, J., « Maintaining knowledge about temporal intervals », *Communications of the ACM*, ACM, vol. 26 (11), New York, NY, USA, 1983, p. 832–843.

[2] Allombert, A., « Aspects temporels d'un système de partitions musicales interactives pour la composition et l'exécution », *Ph.D. Thesis*, Bordeaux, France, 2009.

[3] Allombert, A., Desainte-Catherine, M., and Assayag, G., « Iscore: A system for writing interaction », *Proceedings of the Third International Conference on Digital Interactive Media in Entertainment and Arts*, ACM, New York, NY, USA, 2008, p. 360–367.

[4] Baudart, G., Jacquemard, F., Mandel, L., and Pouzet, M., « A synchronous embedding of Antescofo, a domain-specific language for interactive mixed music », *Proceedings of the Thirteen International Conference on Embedded Software*, Montreal, Canada, 2013.

[5] Baudart, G., Mandel, L., and Pouzet, M., « Programming mixed music in ReactiveML », *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling*, Boston, USA, 2013.

[6] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and De Simone, R., « The synchronous languages 12 years later », *Proceedings of the IEEE*, IEEE, vol. 91 (1), 2003, p. 64–83.

[7] Berry, G., and Gonthier, G., « The Esterel synchronous programming language, design, semantics, implementation », *Science of Computer Programming*, Elsevier, vol. 19 (2), Amsterdam, The Netherlands, 1992, p. 87–152.

[8] Boussinot, F., and De Simone, R., « The SL synchronous language », *IEEE Transactions on Software Engineering*, IEEE Press, vol. 22 (4), Piscataway, USA, 1996, p. 256–266.

[6] `http://scrime.labri.fr`

[9] Cont, A., « ANTESCOFO: Anticipatory synchronization and control of interactive parameters in computer music », *Proceedings of International Computer Music Conference*, Belfast, Ireland, 2008.

[10] Desainte-Catherine, M., Allombert, A., and Assayag, G., « Towards a hybrid temporal paradigm for musical composition and performance: The case of musical interpretation », *Computer Music Journal*, MIT Press, vol. 37 (2), Cambridge, MA, USA, 2013, p. 61–72.

[11] Fober, D., Orlarey, Y., and Letz, S., « An environment for the design of live music scores », *Proceedings of the Linux Audio Conference*, CCRMA, Stanford University, California, US, 2012, p. 47–54.

[12] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D., « The synchronous dataflow programming language LUSTRE », *Proceedings of the IEEE*, IEEE, vol. 79 (9), 1991, p. 1305—1320.

[13] Halbwachs, N., Lagnier, F., and Raymond, P., « Synchronous observers and the verification of reactive systems », *Proceedings of the Third International Conference on Methodology and Software Technology*, Springer-Verlag, London, UK, 1994, p. 83–96.

[14] Halbwachs, N., and Raymond, P., « Validation of synchronous reactive systems: From formal verification to automatic testing », *Proceedings of the Fifth Asian Computing Science Conference on Advances in Computing Science*, Springer-Verlag, 1999, p. 1–12.

[15] Mandel, L., and Plateau, F., « Interactive programming of reactive systems », *Electronic Notes in Theoretical Computer Science*, Elsevier, vol. 238 (1), Amsterdam, The Netherlands, 2009, p. 21–36.

[16] Mandel, L., and Pouzet, M., « ReactiveML, a reactive extension to ML », *Proceedings of the Seventh ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, Lisbon, Portugal, 2005.

[17] Marczak, R., Desainte-Catherine, M., and Allombert, A., « Real-time temporal control of musical processes », *The Third International Conferences on Advances in Multimedia*, Budapest, Hungary, 2011, p. 12–17.

[18] Pouzet, M., « Lucid Synchrone, version 3. Tutorial and reference manual », *http://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf*.

[19] Sénac, P., De Saqui-Sannes, P., and Willrich, R., « Hierarchical Time Stream Petri Net: A model for hypermedia systems », *Proceedings of the Sixteenth International Conference on Application and Theory of Petri Nets*, Springer, Turin, Italy, 1995, p. 451–470.